Student Name:
Chen-yang Yu (862052273) Po-Chang Kuo (862029279)
PART1:
We first modify the following files.
in proc.h add declaration of stackSize

```
struct inode *cwd;           // Current directory
char name[16];               // Process name (debugging)
int stackSize;               //OS153: the size of stack(unit: #page)
```

In defs.h modify pde_t, add argument for stackSize

```
int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
pde_t*          copyuvm(pde_t*, uint, uint);//CS153: add argument for stackSize
void            switchuvm(struct proc*);
```

In order to change the position of stack, we have to

1. Change the stack declare in exec() (change the stack point also) (in exe.c)
   In exe.c, we allocate physical memory for stack starting from the last byte of user memory so let sz = KERNBASE-1

```
  // allocate
  // this is OK:uint new_sp = allocuvm(pgdir, PGROUNDDOWN(KERNBASE-1)-PGSIZE, PGROUNDDOWN(KERNBASE-1));
  uint new_sp = allocuvm(pgdir, PGROUNDDOWN(KERNBASE-1), KERNBASE-1);
  //uint new_sp = allocuvm(pgdir, KERNBASE-1-PGSIZE, KERNBASE-1);
  if (new_sp == 0)
      cprintf("new_sp: fail\n");
  sp = new_sp;
  cprintf("new_sp ok: %x (%d)\n",sp,sp);
```

2. Change the statement which check the boundary address when access stack (in syscall.c sysfile.c)
   In syscall.c, because we put the stack at the top of memory so if any process that would access stack space would report error.

```
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;
  //struct proc *curproc = myproc();

  //OS153if(addr >= curproc->sz)
    //OS153 return -1;
  if(addr >= KERNBASE)
    cprintf("something wrong in fetchstr!!\n");
  *pp = (char*)addr;
  //OS153ep = (char*)curproc->sz;
  ep = (char*)(KERNBASE-1);
  for(s = *pp; s < ep; s++){
    if(*s == 0)
      return s - *pp;
  }
  return -1;
}
```

```
int
fetchint(uint addr, int *ip)
{
  //struct proc *curproc = myproc();

  //OS153 if(addr >= curproc->sz || addr+4 > curproc->sz)
    //OS153 return -1;
  if(addr >= KERNBASE || addr+4 >= KERNBASE)
      cprintf("something wrong in fetchint!!\n");
  *ip = *(int*)(addr);
  return 0;
}

int
argptr(int n, char **pp, int size)
{
  int i;
  //struct proc *curproc = myproc();

  if(argint(n, &i) < 0)
    return -1;
  //OS153if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    //OS153 return -1;
  if(size < 0 || (uint)i >= KERNBASE || (uint)i+size > KERNBASE)
    cprintf("something wrong in argptr!!\n");
  *pp = (char*)i;
  return 0;
}
```

3. When fork() system call, copy the stack in the new position (in vm.c, add a proc->stackSize in proc.h to record the size of stack so we know how many space we need to move)
   In vm.c

```
//OS153: 0:sz only contain the static data, we have to copy stack also
//proc->stackSize store the number of pages we allocate for stack
//uint stackPtr = KERNBASE-1-1*PGSIZE;
//OK:uint stackPtr = PGROUNDDOWN(KERNBASE-1)-PGSIZE;
uint stackPtr = PGROUNDDOWN(KERNBASE-1)-(stackSize-1)*PGSIZE;
//uint stackPtr = stackSize;
//OK:for(i = stackPtr; i <= PGROUNDDOWN(KERNBASE-1); i += PGSIZE){
for(i = stackPtr; i <= KERNBASE-1; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 1)) == 0)
        panic("copyuvm: pte should exist2");
    if(!(*pte & PTE_P))
        panic("copyuvm: page not present2");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
        goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
        goto bad;
}
```

4. When stack grows over a single page, allocate a new page for it (in trap.c, add another page beyond the current stack position, the stack position can be calculate by proc->stackSize)
   In trap.c

```
case T_PGFLT:{
    uint errorAddress = rcr2();
    uint stackTop = PGROUNDDOWN(myproc()->tf->esp);
    if(errorAddress <= PGROUNDUP(myproc()->sz)+PGSIZE){// force stop, already write to data sector
        cprintf("force stop, already write to data sector\n");
        cprintf("rcr2()=0x%x, proc->sz=0x%x, stackSize=%d\n",errorAddress,myproc()->sz,myproc()->stackSize);
        myproc()->killed = 1;
    }else{
      if((errorAddress < stackTop) && (errorAddress > stackTop-PGSIZE)){
        if(allocuvm(myproc()->pgdir, stackTop-PGSIZE, stackTop)==0)
            freevm(myproc()->pgdir);
        myproc()->tf->esp -= PGSIZE;
        //cprintf("allocc succ\n");
      }
    }
    if(myproc()->tf->esp<0x100000)
        cprintf("sp:0x%x\n",myproc()->tf->esp);
    break;
}
```

BONUS:
Turn out if we places the stack in the end of user space, it is impossible to grow into heap (assume heap is not really big). The limitation is 224 MB for our stack. This limitation shows when we use the test program provided by professor. Since we don't set the terminal condition, the program will keep allocate page for stack, until it shows this:

```
init: starting sh
OS153: sz = 0x2950 (10576)
new_sp ok: 7fffffff (2147483647)
OS153: sp = 0x7ffffe8 (2147483624)
$ test
OS153: sz = 0x1a64 (6756)
new_sp ok: 7fffffff (2147483647)
OS153: sp = 0x7ffffe4 (2147483620)
allocuvm out of memory, in a:0x722a9000
EAX=01010101 EBX=01000000 ECX=00000400 EDX=8dff7000
ESI=8df72000 EDI=8dff7000 EBP=8dfbcf08 ESP=8dfbcf00
EIP=8010445c EFL=00000012 [----A--] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0000 00000000 00000000 00000000
GS =0000 00000000 00000000 00000000
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0028 80112788 00000067 00408900 DPL=0 TSS32-avl
GDT=     801127f0 0000002f
IDT=     80114da0 000007ff
CR0=80010011 CR2=8dff7000 CR3=0df72000 CR4=00000010
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
QEMU: Terminated
```

The reason is: although the address space allowed to use is 2GB, the actual physical memory space allowed to allocate is 224 MB. This limitation is set as PHYSTOP (define in memlayout.h), which is 0xE000000 = 224*1024*1024 = 224 MB. The legal page we can use will be free at initial phase.

In main.c:

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  shminit();       // shared memory
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

kinit1() allocate the first 4 MB address space for BIOS, kernel static data
kinit2() allocate the first 224 MB address space for free memory

Turn out the free address space is from 4 MB to 224 MB. When it translate into virtual address, simply add KERNBASE to it. The reason why xv6 set PHYSTOP as 224 MB is xv6 need to map all available address space at initial phase. If we set it as 2GB, it will take too long to map all pages. However, we can still modify this value if we expect a large program will run on system.

PART2:
The mechanism of memory sharing is given and we only need to implement the method(shm_open() and shm_close(), both system call located in shm.c)
In shm_open(int id, char ** pointer):

For a specific id in shm_table, we have to return a virtual memory address, which is mapping to a physical address. Since this physical address can be access through other process, we actually share this segment of memory between different process. We return this address by the second parameter (char ** pointer).

Case 1: When id is found: We have a specific physical address which is saved in shm_table, just map it to our virtual address and return

a. Use mappages() to map the frame(physical page) to virtual address:
Go through shm_table, if we find the right id, we use the *frame to map a virtual address in our memory space. The virtual address is PGROUNDUP(our current memory space), which is the next multiple of page size.

```
for(i=0;i<64;i++){
        if(shm_table.shm_pages[i].id == id){
    //hit
    cprintf("HIT!!\n");
    found = 1;
    acquire(&(shm_table.lock));
                shm_table.shm_pages[i].refcnt++;
    release(&(shm_table.lock));
    //mapping
    //char *va = PGROUNDUP(myproc()->sz);
    cprintf("HIT2!!\n");
    uint va1 = PGROUNDUP(myproc()->sz);
    if(mappages(myproc()->pgdir, (char*)va1, PGSIZE, V2P
(shm_table.shm_pages[i].frame), PTE_W|PTE_U) < 0){
        cprintf("allocuvm out of memory (2)\n");
    }
    //return this va
```

b. Return this virtual address by storing to input parameter and update the proc->sz (we use a page, this have to be calculated into memory space)

```
//return this va
*pointer = (char *)va1;
//update sz
myproc()->sz = PGROUNDUP(myproc()->sz);
```

Case 2: When id is not found: We have to allocate a physical page and add to shm_table, which is the real address we will put our share data)

a. Create an empty element(indicate the refcnt is zero, no process is using it) in shm_table and put id

```
if(shm_table.shm_pages[i].refcnt == 0){
    cprintf("NEW!!!\n");
    //inc refcnt, assign id
    acquire(&(shm_table.lock));
      shm_table.shm_pages[i].refcnt++;
      shm_table.shm_pages[i].id = id;
    release(&(shm_table.lock));
```

b. Allocate a physical memory page and put it in shm_table
This is similar with the function allocuvm(). We use kalloc() to allocate a physical page, the return value is the address of the beginning of the physical page. And put this address into shm_table.frame (after initialize it to zero). This is the physical segment we would like to put shares data.

```
                //allocate a page and save to *frame
                mem = kalloc();
                cprintf("NEW2!!!\n");
                if(mem == 0){
                    cprintf("allocuvm out of memory\n");
                    //deallocuvm(pgdir, newsz, oldsz);
                    //return 0;
                }else{
                    memset(mem, 0, PGSIZE);
                    acquire(&(shm_table.lock));
                      shm_table.shm_pages[i].frame = mem;
                    release(&(shm_table.lock));
                }
```

c. Map this new physical page to virtual address
   Since we have a segment which can put shared data. The rest is the same as case1. Map this segment to our memory space.

```
                //have a segment in table, mappage() to it
                uint va2 = PGROUNDUP(myproc()->sz);
                if(mappages(myproc()->pgdir, (char*)va2, PGSIZE, V2P
(shm_table.shm_pages[i].frame), PTE_W|PTE_U) < 0){
                    cprintf("allocuvm out of memory (2)\n");
                }
                cprintf("NEW3!!!\n");
                //return this va
                *pointer = (char *)va2;
                //update sz
                myproc()->sz = PGROUNDUP(myproc()->sz);
                //release(&(shm_table.lock));
                //break;
```

In shm_close(int id):

It means this process stops the sharing, find the corresponding id in shm_table and subtract the reference count. Go through the table and modify the refcnt value which id is the same as input parameter.

```
int shm_close(int id) {
#if 1
    //you write this too!
        for(uint i=0;i<64;i++){
                if(shm_table.shm_pages[i].id == id){
            //hit
            acquire(&(shm_table.lock));
                        shm_table.shm_pages[i].refcnt--;
            release(&(shm_table.lock));
            break;
                }
        }
#endif
    return 0; //added to remove compiler warning -- you should decide
what to return
}
```

The result:

Test program is provided. The original counter should be increased by two process 10000 time separately. However, due to both process increase the value at the same address, one of the final result would be 10000+10000, which is 20000.

```
$ shm_cnt
NEW!!!
NEW2!!!
NEW3!!!
Counter in Parent is 1HIT!!
HIT2!!
HIT3!!
Co at address 0x4000
unter in Child is 2 at address 0x4000
Counter in Child is 1002 at address 0x4000
CCounter in Parent is 3002 at address 0x4000
Counter in Parent is 4002 at address 0x4000
ounter in Child is 2002 at address 0x4000
Counter in Child is 6002 at address 0x4000
Counter in Child is 7002 at address 0x4000
CCounter in Parent is 5002 at address 0x4000
Counter in Parent is 9002 at address 0x4000
Counter in Parent is 10002 at address 0x4000
Counteounter in Child is 8002r in Parent is 11002 at address 0x4000
Counter in Parent is 12002 at address 0x4000
Counte at address 0x4000
Counter in Child is 14002 at address 0x4000
Counter in Child is 15002 at address 0x4000
Counter in Child is 16002 at r in Parent is 13002 at address 0x4000
Counter in Parent is 17002 at address 0x4000
Counter in parent is 18001
address 0x4000
Counter in Child is 19001 at address 0x4000
Counter in child is 20000
```